

Rhys Advanced Software Development Exam

Section A

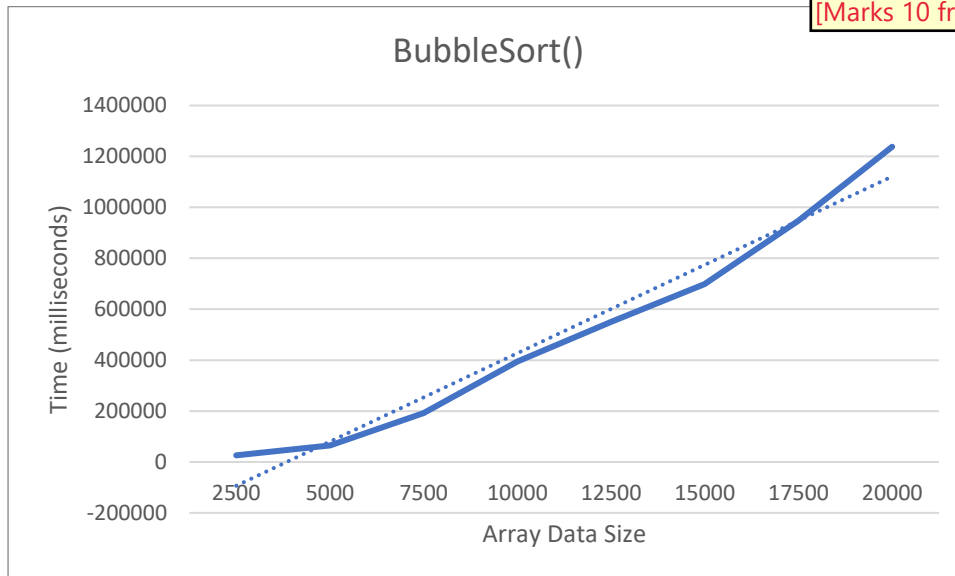
Question 1

A

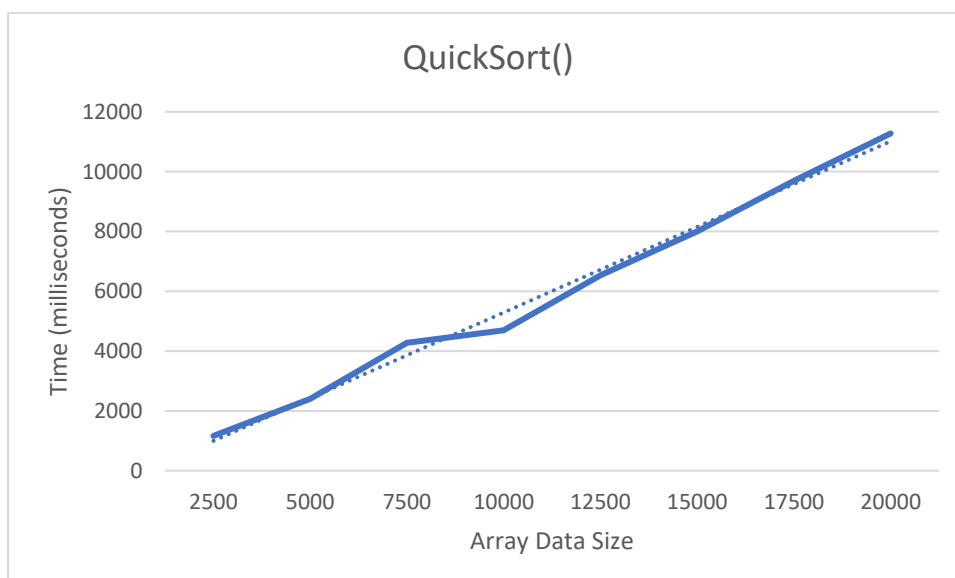
Q1.a

Question answered fully

[Marks 10 from 10]



The BubbleSort() trendline shows as a straight line, but the data line looks more like a curve to me, where it starts to take more time as the data set increases. This points towards a big O of n^2 , or a polynomial complexity.



The trendline for QuickSort() is also a straight line, however the data is quite straight too, with a slight irregularity between 7500 and 10000 items of data. I believe this is a linear complexity, or n . The trendline on these graphs is the blue dotted line, where the solid blue line is the data.

B

Bubble sort algorithm – I think it is a polynomial function (n^2) because it has a for loop within the while loop. Having a loop within a loop in a sorting algorithm typically causes it to get slower with more data used.

Quick sort partition – Quick sort sorting algorithms typically have a range of efficiency depending on how the data is sorted. As it is a quick sort algorithm, at its best it would likely be $\log n$ efficiency, but at its worst it could be a n^2 just like the bubble sort. This one is quite a large implementation, with quite a few while loops and variable changes. I'd guess that this is a fairly slow quick sort function, but I still think it has the potential to be fast in some circumstances.

Quick sort - Quick sort sorting algorithms typically have a range of efficiency depending on how the data is sorted. As it is a quick sort algorithm, at its best it would likely be $\log n$ efficiency, but at its worst it could be a n^2 just like the bubble sort. This one is quite small and simple; it looks at the two data entries at the start and end of the database and moves them around if they aren't in order. This function would likely run many times to fully sort the data. Its performance is going to be quite similar to the other quick sort function, but with so little to it, I can imagine it would be a bit faster.

C

Q1.b
 Very good answer
 [Marks 6 from 7]

1 – some algorithms need to repeat multiple times to be able to get the desired result from a program. They might only carry out one or a few simple steps each time the code is ran, whereby the steps taken might have to be repeated over and over. In programming, this is called recursion, which is typically achieved with loops and a function call within the loop. A loop would consist of a condition which has to be met in order for the loop to stop repeating. The code which then wants to be repeated would be contained within the loop, or the function call to the desired code. Two examples of loops could be for loops and while loops. A for loop repeats for a specific number of times, while a while loop would repeat indefinitely until its criteria has been met. For the partition quick sort, it has while loops within its code, repeating operations until the job has been done this is not an example of recursion as it is not calling on a function. As for the other quick sort, its loop would exist elsewhere, calling the function over and over again until the criteria has been met. This is an example of recursion.

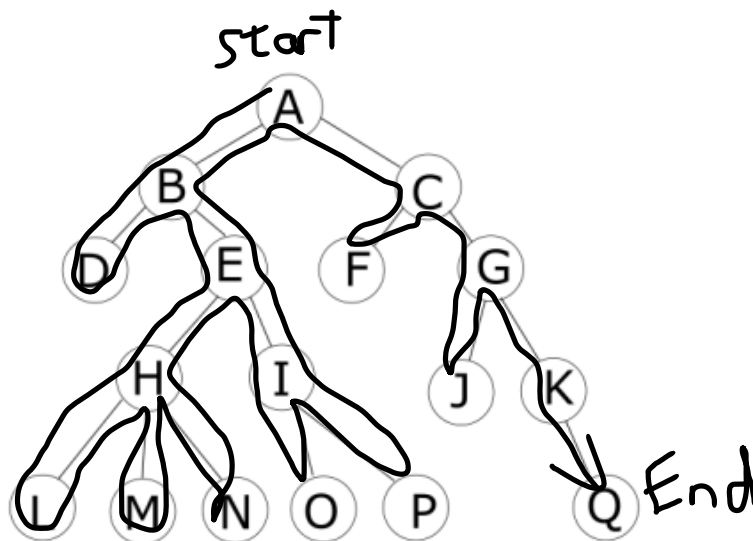
2 – the final function called the quicksort algorithm would be an example of recursion. A loop would exist elsewhere (such as in the main or in another function) which would be calling this function continuously until the loops conditions have been met.

```
FUNCTION quickSort (array, start, end): IF start < end THEN SET pivot TO partition(array, start, end)
quickSort(array, start, pivot -1) quickSort(array, pivot + 1, end) RETURN array END FUNCTION
```

Section B

Question 2

A



Q2.a

Very good answer covering the key points of a DFS.

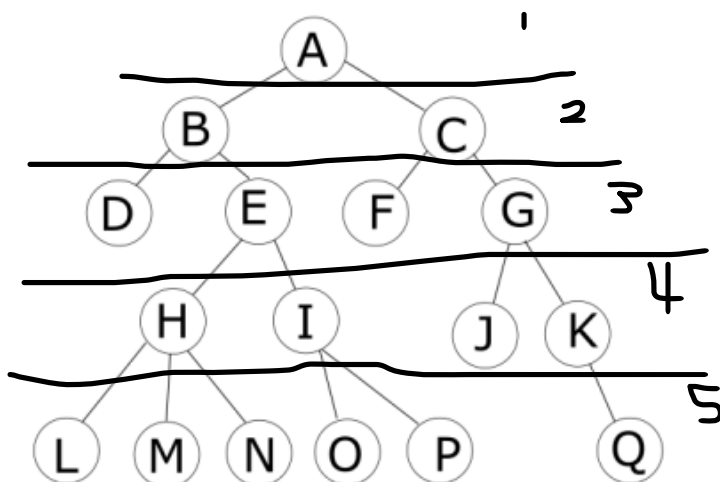
[Marks 9 from 9]

List of node visits from first to last: A (root), B, D, B, E, H, I, H, M, H, N, H, E, I, O, I, P, I, E, B, A, C, F, C, G, J, G, K, Q

The image above shows how I think the depth first search would move through the graph. A depth first search will travel down one branch as far as possible, then head back to search the next branch. I assumed here that the search will always prioritise the left most branch first.

This type of search can take quite a while depending on where the desired node is, which way the algorithm is designed to go and the layout of the graph. It will also have to remember where it has been in the graph to ensure it doesn't search the same branches again, which could cause it to get stuck in an infinite loop of searching. If made correctly, the algorithm should always find the end, making it a decent method for maze searching, but could take a really long time with many different paths, which is likely why this type of search is not used in GPS and networking systems.

B



Q2.b

Another good answer with just minor omissions

[Marks 7 from 10]

Queue at part 1: B, C

Queue at part 2: D, E, F, G

Queue at part 3: H, I, J, K

Queue at part 4: L, M, N, O, P, Q

Queue at part 5: NULL

You would likely use a queue and a list to implement a depth first search. You would have the queue to store the locations which you need to visit next, and a list to hold the nodes which have been visited. You would then use while loops to add nodes to the list and remove nodes from the list (push and pop) as you are searching new locations. For example, when b is searched, d and e get added to the queue and b gets removed from the queue to the list.

C

Pushing to a queue adds an item to the end of the queue, popping an item removes the first item in the queue. Sometimes more or less items get pushed or popped compared to the other operation, so the list should have that considered to continue to function properly. This can be done by ensuring the list is long enough and checking the last value instead of the last space in the queue, as the spaces could be left empty if less items are in the queue.

Question 3

A

1 first to last – 77 (root), 40, 39, 42, 86, 85, 90

2 first to last – 39 (start), 42, 40, 85, 90, 86, 77 (root)

3 first to last – 39 (start), 40, 42, 77, 85, 86, 90

2.c

Good answer

[Marks 4 from 6]

Q3.a

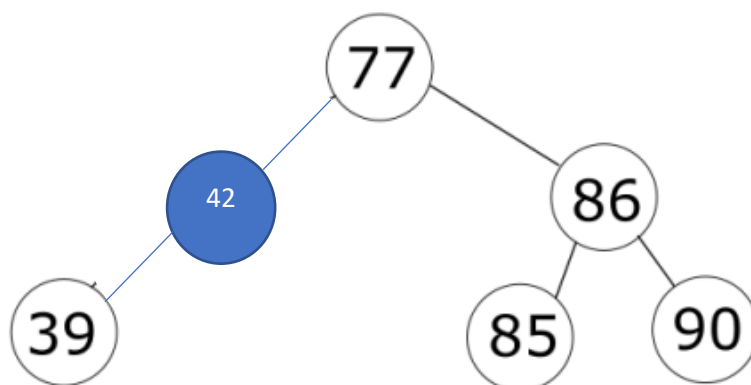
The order of nodes correctly identified for each traversal method.

[Marks 12 from 12]

B

Adding 65 to the table above. 65 is smaller than 77, which means that the 65 node would definitely be placed on the left branch. It would then get to 40, whereby 65 is bigger than 40, so it will move down through the right edge. When it gets to 42, it would know that 65 is bigger than 42, but see that there are no edges connected to 42. It would then create a new edge going down to the right of the 42 node, where it will create a new node for 65 attached to that edge. I'd say this would probably end up being a linear category big O as adding more nodes to this tree would likely just make this searching process one step longer with each row it has to travel.

C



Q3.b

Correctly discussed the insert process and a brief analyse of the operation was given

[Marks 6 from 8]

The 40 node would get removed, whereby the 42 node which is the next closest node on the left branch to the root node would take its place and connect to the 39 and root nodes. This can be seen in the image above.

Q3.c

The tree diagram has been correctly reproduced and explained

[Marks 5 from 5]

Summary:

Question 1: 24

Question 2: 20

Question 3: 23

Question 4:

Total: 67 from 75 (89%)